

C/C++ Memory Model and Interactions

Project 1

Alex Harper

Contents

1	Introduction	1
1.1	Multi-Tasking	1
2	MPI	2
2.1	Time Comparisons	4
2.2	MPI on Multiple Systems	5
3	Pthreads - The Old Workhorse	5
3.1	v2	8
4	OpenMP	9
A	reference.cpp	13
B	mpi.cpp	16
C	pthread2.cpp	24
D	openmp.cpp	30

List of Figures

1	Book Hello World	2
2	Having Processes Race to Print to stdout	2
3	Process Diagram For Mergeing Data Down	3
4	Sorting Time in Microseconds for 100 Million Items	4
5	Multi-Threaded Recursion ; Top-Original, Bottom-Threaded	6
6	Psudo Code For Pthread	6
7	Psudo Code For Pthread Attempt 2	7
8	Psudo Code For Pthread Attempt 2 - Worker	7
9	Sorting Time in Microseconds for 100 Million Items	8
10	Sorting Time in Microseconds for 100 Million Items	9
11	malloc() Corruption Issue Somehow	9
12	Psudo Code For Pthread Attempt 2 - Worker	11
13	Sorting Time in Microseconds for 100 Million Items	12

1 Introduction

This paper is about the 3 main things the book covers for accomplishing multi-tasking. Each has its strengths and weaknesses for different tasks, and each has its own variety of pitfalls.

I have familiarity with multi-threading because some tasks (like GUIs) greatly benefit from splitting things up. In the past I have used pthreads directly before, but did more of a fire and forget way. I have also used some builtin things like for python or the Qt threading stuff. Just in the more recent years I have started making use of even passing data between threads. Working through these different parts with my bit of experience was interesting to me.

1.1 Multi-Tasking

While I admit I only skimmed through the beginning of the book, I believe that it is not entirely clear about the two main constructs that we are concerned with. The two things are multi-threading and multi-processing. The book covers the concept of having shared or separate memory in fair detail. It also covers the different levels of things running simultaneously. But what I did not notice in the book was the direct mentioning of where what things are what.

Multi-threading is when you have a shared memory space but still asking for multiple time slices at the same time. This is what pthreads and OpenMP do, where you can have multiple instruction streams all sharing the same memory. This is often used because it is cheap to do; no memory isolation, no weird copy-on-write for shared memory, and just simpler.

Multi-processing is when you have multiple processes, each with their own memory space. This is better for isolation, often can even be setup so one process dying does not crash everything else. This though is more heavy of an operation, where the OS has to setup protections again for every new process. It also forces you to use dedicated mechanisms to have different processes talk to each other. MPI is this form of multi-tasking.

2 MPI

MPI is a multi-process framework. It focuses on spawning many processes and then having messages passed between them to coordinate work. The bulk of the framework's use is the automated transport of messages between processes. The messages can be anything, and what you put in the messages is up to you.

So as all programming starts out, there is a “Hello World” program. I typed in the book's version to make sure my install of things was working. As is obvious, it works just fine, and prints the normal stuff. 1

Code	<pre>1 //pretend there is code here 2 //I think it is silly to put the example code in the 3 //book into the report</pre>
Output	<pre>Master process of 4 processes Process 1 of 4 Process 2 of 4 Process 3 of 4</pre>

Figure 1: Book Hello World

So, just to get the obligatory multi-tasking concept out of the way, I changed the code to print from the child processes instead of the master process. This should be familiar territory where the sequence of prints is unpredictable. Funnily, it took me running the same program several times before it swapped anything. 2

Code	<pre>1 //Psuedo Code to make reading easier 2 if(master_process){ 3 printf("Master process of %d processes\n", num_procs); 4 for(others=1; others<num_procs; others++) 5 MPI_Send("Process %d of %d",others); 6 } else { 7 MPI_Recv(buffer,master_process); 8 printf("%s\n",buffer); 9 }</pre>
Output	<pre>Master process of 4 processes Process 2 of 4 Process 1 of 4 Process 3 of 4</pre>

Figure 2: Having Processes Race to Print to stdout

So with the standard set of things out of the way, lets do something more interesting. I am thinking a distributed merge sort will be challenging enough. So the first step of any project that is going to multi-task is planning the division of work.

Merge sort works by splitting all the data into halves until there are only 2 items. The two items are returned sorted (eg the smaller then the bigger item). Then it is combined with the next piece, where you interleave the 4 items into a single list to pass up the chain. You keep passing up the chain until you interleave all the items in a final merge.

My starting thought was that each process should get a number of pairs of base data to get the process started. After that, how to percolate the data up is the question. There are 3 options for what to do with the data already sent; send it back to the master, send it to intermediate processes, or keep it where it is. The results being sent directly to the master is wasteful because not enough work has happened on each node yet. If sent to intermediate nodes, I'm not sure what to do for the nodes that don't have work to do. But keeping the data where it is, we could send more data to it until we have emptied our list.

Since I like the last option, I thought about it and refined it down. Instead of dolling out a couple items at a time, it would be better to send a whole chunk at once. That sounds something like the book mentioned; `MPI_Scatter`. The problem with that function though, is if the number of items does not evenly divide by the number of processes, we will have items left over in our list. With the leftovers, they can either be handled by the master, or a second recieve could be done to try to push out the work out. I don't think that it would be a large number of items ever left with the master thread if we just do cleanup with it later. If 7 threads and 10k items, then only 4 items would be left over.

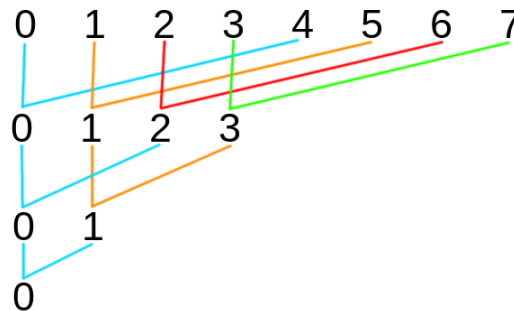


Figure 3: Process Diagram For Merging Data Down

So this is my final plan before I have even written any code for it. The master who has the list will break it up and send sections to the worker threads using

MPI.Scatter. When each worker has finished their sorting, they send it to another node using the tree pattern like what MPI.Reduce might use. Using this, the sorted list eventually works it's way back up to the master process who will do the final merge of the sorted parts. You can see the tree reduction in fig3 for getting back to the master thread.

2.1 Time Comparisons

	Time
MPI (4 threads)	5716330
MPI (8 threads)	8120605
MPI (2 threads)	10940588
Reference	17910059
MPI (4 threads,optimised)	5567745
Reference (optimised)	10480717

Figure 4: Sorting Time in Microseconds for 100 Million Items

So after getting things done, the version using MPI on only my local system is faster! This is good news that spreading it across 4 cores does help, because I feel vindicated as to my skill of splitting tasks up. This test for having only 4 threads (one per physical core) on my machine is the fastest which makes sense because it can use every core to it's fullest without stepping on other processes. I think that it does not scale with more processes because my machine does not have enough hardware for it. But looking at the optimal multi-process, it does not scale perfectly. If you multiply the time by 4, it is 22 seconds, where the reference only took 17 seconds. This shows that there is overhead still in play, but is worth it to split the task up.

One thing that is interesting is when we start trying to pass optimizer flags to the compiler. Doing the normal level optimizing, the reference code gets a fair bit faster. But the same flags for the MPI version sees very little improvement. I have no clue what that means, but with this, the gap is seriously reduced. With this smaller gap, the benefit is diminished, and so would require even larger projects to justify the extra coding time MPI takes.

I should mention a caveat with my code; it only scales to powers of 2 for processes. The merging during the sort makes the lowest order processes take data from the higher orders. While this is great for being a distributed work load for the merge, it does mean that sometimes a stragler process is left out. If I had more time, I could try implementing an idea I have for leftover process that don't even divide at every step, but for testing, the code would not really be different enough to matter.

2.2 MPI on Multiple Systems

So all is well and good for running on the same computer, but if that was all we were going to do, we might as well use a lighter weight multi-threading setup. Where MPI is very useful is having processes on different systems communicate over a fast interlink. I took a look online for how to do it and the first link was this source.

So it seems that you have to have a dedicated list of IPs for every machine you want it to run on. This seems fine for a static environment like a super computer, but I don't think it is capable of dynamically changing the number of processes. This means that it is limited to some degree, and requires knowing and managing the resources you need before running. I could be wrong, but there are too many functions on the reference page for MPI.

3 Pthreads - The Old Workhorse

Since the early days of unix, simple APIs for doing things have been a staple. Pthreads comes from this early time, and with their simple function calls allow programs to do multi-threading.

To use these in this paper, I am (again) going to try making a merge sort. So my thoughts on how to do this; I have my reference program from the previous section that does all the tricky bits, so I will just modify that. My first attempt was to intercept where I was splitting the list into smaller pieces recursively and simply make each split a new thread. You can see the simple change I made in fig5 where a single line turns into several.

It works by simply wrapping my original function that recursively divides the data called "merge_sort_recurse". The original function would split the data in half, check the size, split in half, check the size, until there were only 2 items. Then it comes back up and does the merging that gives the algorithm its name. So the minor change was to have it make a new thread everytime it split the data in half.

This worked just fine when I did my first test on a small list of only 10 elements. But when I asked it to do real work, it simply segfaults. The tipping point seems to be going from 10k to 100k items in the list, where there are just too many threads spawned for my computer to handle. So that means this technique does not scale well enough for my taste and I need to try something else.

So perhaps the way to do it is spawn a set number of worker threads and then throw work at them. This will not be the same as the MPI system where each

```

1  for(every half of data){
2      //recursively cut down the data
3      bool swap_left =
4      merge_sort_recurse(buf,half,work);
5  }


---


1  void* merge_sort_recurse_wrapper(void* param_pointer){
2      Params* param = (Params*) param_pointer;
3      *(param->return_value) = merge_sort_recurse(
4          param->buf,param->size,param->work);
5      return NULL;
6  }
7  for(every half of data){
8      Params left={&swap_left,buf,half,work}
9      pthread_t handle;
10     pthread_create(handles,NULL,merge_sort_recurse_wrapper,&left);
11     pthread_join(handles[0],NULL);
12 }

```

Figure 5: Multi-Threaded Recursion ; Top-Original, Bottom-Threaded

thread is given all the data and when they finish they die off. Instead I am thinking of posting jobs to a global list somehow and each thread will take a job to do on it's own. The concept is mostly summed up in fig6 even though it will require a lot more in the way of specifics.

<pre> 1 while(num_threads < wanted_number) 2 start_worker_thread(); 3 for(every section of data) 4 add_data_section_to_list(section); </pre>
<pre> 1 void worker_thread(){ 2 while(have_data_segments){ 3 data_segment = get_first_item_off_list(); 4 sort(data_segment); 5 } 6 } </pre>

Figure 6: Psudo Code For Pthread

So making a global work queue, what does that look like? There will be a global variable to point to the object, there will be a mutex to keep things sane. Seems fairly easy if I just make a linked list with the parameters I need. You can see a sortend version of it in fig7.

```

1 struct list_item{list_item* next;int *buf,size,*work;};
2 pthread_mutex_t queue_mutex;
3 struct list_item *list_front=NULL,*list_back=NULL;
4 void add_work_item(int*buf,int size,int*work);
5 struct list_item* get_next_work(){
6     pthread_mutex_lock(&queue_mutex);
7     //manage the list
8     pthread_mutex_unlock(&queue_mutex);
9     return work_item;
10 }

```

Figure 7: Psudo Code For Pthread Attempt 2

And then for the worker thread, it is fairly simple for shuffling data around. Ask for work, make sure you actually got some, sort the item or exit. fig8

```

1 void* worker_thread(void*){
2     auto segment = get_next_work();
3     while(segment != NULL){
4         merge_sort_sort(segment);
5         segment = get_next_work();
6     }
7 }
8 pthread_t handles[NUM_THREADS];
9 for(int x=0; x<NUM_THREADS; x++)
10     pthread_create(handles+x,NULL,worker_thread,NULL);
11 for(int x=0; x<NUM_THREADS; x++)
12     pthread_join(handles[x],NULL);

```

Figure 8: Psudo Code For Pthread Attempt 2 - Worker

So with me implementing in code, how does it compare? The gory numbers are show in fig9. The way I am doing the sorting is apparently very terrible, with more threads just making things worse.

I believe that the problem comes from the way I am passing out work. There is a mutex to lock for every time a thread wants to get more work to do, which is an expensive operation. I would need to fundamentally change the way my program works if I want to correct this problem. I likely would try adding a job queue to every thread instead so that there are multiple contention points, and not just a single one.

There is possibly another problem that I don't think is much of the stall

	Time
MPI (4 threads)	5716330
Pthreads(1 threads)	17771804
Reference	17910059
Pthreads(2 threads)	29780136
Pthreads(4 threads)	32644114
Pthreads(8 threads)	36312550
MPI (4 threads,optimised)	5567745
Reference (optimised)	10480717
Pthread (2 threads,optimised)	26507091

Figure 9: Sorting Time in Microseconds for 100 Million Items

time. Since it must merge groups together, there is a dependency for some jobs to use data from other jobs that should already be calculated. I did not separate the jobs out quite right, so there is likely some threads waiting for other to finish first.

But alas, I am out of time to work on this, and we will see if things work out any better in the next section.

3.1 v2

So I HAD to make this work in a different manner, which was a fairly large code change. The difference is that the setup of the threads is no longer recursive. There are some benefits to my new code.

- Can use the source to do the OpenMP section because I now have loops
- Slightly faster even though the work queue is almost exactly the same
- The sorting now gives back correct results
- No weird dependency graph as much as before because things are stacked better this time
- Works for ANY number of threads, even prime numbers!

work, there are none. This has been my problem with the pthreads, no easy loops to work with because I made everything recursive.

So there are 3 options; make the program use loops instead of recursion, do another topic, give up because the paper is due in a couple hours. While I seriously considered the last option, there is still time to try the others. I spun the gears in my head to come up with a new idea to work on, but my creativity does not work that way. Seems I'm stuck working with what I have.

After having modified the pthreads version to be non-recursive (and other benefits), it should be easier to try out OpenMP. Like before, I look for any loops that I can directly add OpenMP onto. Unfortunately, the only for loops are in the queue init function and have lost of dependencies on previous state. That means I am stuck with looking only at the threads coming off the work queue. Luckily it was fairly simple to convert the pthread code to the openmp code.

Fig12 shows the comparison of the different codes. The only major change is how I define the mutex for getting work from the queue. The mutex used to be buried in the `get_work()` function, but now I am specifying the assignment as critical. The barrier condition is a little funny looking now, but it seems to work. Starting the various threads is shorter and nicer looking.

```

1 void* worker_thread(void*){
2     auto segment = get_next_work();
3     while(segment != NULL){
4         if(segment->left!=NULL)
5             merge_sort_merge(segment);
6         else // sync command
7             pthread_barrier_wait(&sync_barrier);
8         segment = get_next_work();
9     }
10 }
11 pthread_t handles[NUM_THREADS];
12 for(int x=0; x<NUM_THREADS; x++)
13     pthread_create(handles+x,NULL,worker_thread,NULL);
14 for(int x=0; x<NUM_THREADS; x++)
15     pthread_join(handles[x],NULL);

1 void* worker_thread(void*){
2     #pragma omp critical
3     auto segment = get_next_work();
4     while(segment != NULL){
5         if(segment->left!=NULL)
6             merge_sort_merge(segment);
7         else{ // sync command
8             #pragma omp barrier
9             ;}
10         #pragma omp critical
11         segment = get_next_work();
12     }
13 }
14 #pragma omp parallel num_threads(NUM_THREADS)
15 worker_thread(NULL);

```

Figure 12: Psudo Code For Pthread Attempt 2 - Worker

Lets look at the times now in fig13. Since the code is almost exactly the pthread code, it makes sense that it is in the same ballpark. What I find interesting though is how there seems to be less variance in the times between the number of threads. There was so little variance, that it took 32 threads before there was more than margin of error, and then it went down in time. (Honestly no clue as to what it is doing) The optimised times show a less drastic improvment that the pthreads, but there still is some.

	Time
MPI (4 threads)	5716330
Reference	17910059
OpenMP(32 threads)	19705202
OpenMP (8 threads)	20043185
OpenMP (2 threads)	20158438
OpenMP(16 threads)	20272851
OpenMP (4 threads)	20741977
OpenMP (1 threads)	20845935
Pthreads(4 threads)	21312523
MPI (4 threads,optimised)	5567745
Reference (optimised)	10480717
OpenMP (2 threads,optimised)	16650686
OpenMP (8 threads,optimised)	17425908
Pthread (4 threads,optimised)	17753379
OpenMP (4 threads,optimised)	18311526

Figure 13: Sorting Time in Microseconds for 100 Million Items

A reference.cpp

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <unistd.h>
5  #include <sys/time.h>
6
7  void merge_sort();
8
9  int main(){
10     merge_sort();
11     return 0;
12 }
13
14
15 #define LIST_SIZE 100000000
16 bool merge_sort_recurse(int* buf,int size,int* work);
17 void merge_sort_merge(int* left_buf,int left_size,int* right_buf,int right_size,int*
18 void merge_sort(){
19     struct timeval timecheck;
20     long start_time,end_time;
21     srand(time(NULL));
22
23     int* buf = (int*)malloc(LIST_SIZE*sizeof(int));
24     int* work = (int*)malloc(LIST_SIZE*sizeof(int));
25     for(int i=0;i<LIST_SIZE;i++)
26         buf[i] = (rand()/(double)(RAND_MAX)) * LIST_SIZE;
27
28     gettimeofday(&timecheck, NULL);
29     start_time = (long)timecheck.tv_sec * 1000000 + (long)timecheck.tv_usec;
30     bool is_backwards =
31     merge_sort_recurse(buf,LIST_SIZE,work);
32     gettimeofday(&timecheck, NULL);
33     end_time = (long)timecheck.tv_sec * 1000000 + (long)timecheck.tv_usec;
34
35     // for(int x=0; x<LIST_SIZE; x++)
36     //     if(is_backwards)
37     //         printf("%d  ", work[x]);
38     //     else
39     //         printf("%d  ", buf[x]);
```



```

40     // printf("\n");
41     printf("Time Taken in us : %ld\n",end_time - start_time);
42     free(buf);
43     free(work);
44 }
45 bool merge_sort_recurse(int* buf,int size,int* work){
46     if(size==1){
47         // *work = *buf;
48         return false;
49     }
50     if(size==2){
51         if(buf[0] > buf[1]){
52             int temp = buf[1];
53             buf[0] = buf[1];
54             buf[1] = temp;
55         }
56         // work[0] = buf[0];
57         // work[1] = buf[1];
58         //printf(" %d %d\n\n",work[0],work[1]);
59         return false;
60     }
61     // recursive part
62     int half = size/2;
63     bool swap_left=false,swap_right=false;
64     swap_left =
65     merge_sort_recurse(buf,half,work);
66     swap_right =
67     merge_sort_recurse(buf+half,size-half,work+half); // subtraction for odd siz
68     if(swap_left != swap_right){
69         if(swap_right)
70             for(int i=0; i<half; i++)
71                 work[i] = buf[i];
72         else
73             for(int i=0; i<half; i++)
74                 buf[i] = work[i];
75     }
76 }
77 if(swap_right){
78     int* temp = work;
79     work = buf;
80     buf = temp;
81 }

```

```

82     merge_sort_merge(buf, half, buf+half, size-half, work);
83
84     return !swap_right;
85 }
86 void merge_sort_merge(int* left_buf, int left_size, int* right_buf, int right_size, int*
87     while(left_size && right_size){
88         if(*left_buf <= *right_buf){
89             *dest = *left_buf;
90             left_size--;
91             left_buf++;
92         }else{
93             *dest = *right_buf;
94             right_size--;
95             right_buf++;
96         }
97         dest++;
98     }
99     while(left_size){
100         *dest = *left_buf;
101         left_size--;
102         left_buf++;
103         dest++;
104     }
105     while(right_size){
106         *dest = *right_buf;
107         right_size--;
108         right_buf++;
109         dest++;
110     }
111
112     return;
113 }

```

B mpi.cpp

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <mpi.h>
4  #include <sys/time.h>
5
6  void hello_world_wikipedia();
7  void hello_world_book();
8  void hello_world_masterOnlySend();
9  void merge_sort();
10
11 int main(int argc, char **argv)
12 {
13     //hello_world_wikipedia();
14     //hello_world_book();
15     //hello_world_masterOnlySend();
16     merge_sort();
17     return 0;
18 }
19
20 void hello_world_wikipedia(){
21     char buf[256];
22     int my_rank, num_procs;
23
24     /* Initialize the infrastructure necessary for communication */
25     MPI_Init(NULL, NULL);
26     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
27     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
28
29     if (my_rank == 0) {
30         int other_rank;
31         printf("We have %i processes.\n", num_procs);
32
33         /* Send messages to all other processes */
34         for (other_rank = 1; other_rank < num_procs; other_rank++){
35             sprintf(buf, "Hello %i!", other_rank);
36             MPI_Send(buf, sizeof(buf), MPI_CHAR, other_rank, 0, MPI_COMM_WORLD);
37         }
38
39         /* Receive messages from all other process */
```

```

40     for (other_rank = 1; other_rank < num_procs; other_rank++){
41         MPI_Recv(buf, sizeof(buf), MPI_CHAR, other_rank, 0, MPI_COMM_WORLD, MPI_S
42         printf("%s\n", buf);
43     }
44
45 }else{
46
47     /* Receive message from process #0 */
48     MPI_Recv(buf, sizeof(buf), MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
49
50     /* Send message to process #0 */
51     sprintf(buf, "Process %i reporting for duty.", my_rank);
52     MPI_Send(buf, sizeof(buf), MPI_CHAR, 0, 0, MPI_COMM_WORLD);
53
54 }
55
56 /* Tear down the communication infrastructure */
57 MPI_Finalize();
58 }
59
60 void hello_world_masterOnlySend(){
61     char buf[256];
62     int my_rank, num_procs;
63
64     /* Initialize the infrastructure necessary for communication */
65     MPI_Init(NULL, NULL);
66     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
67     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
68
69     if (my_rank == 0) {
70         printf("Master process of %d processes\n", num_procs);
71
72         for (int other_rank = 1; other_rank < num_procs; other_rank++){
73             sprintf(buf, "Process %d of %d", other_rank, num_procs);
74             MPI_Send(buf, strlen(buf)+1, MPI_CHAR, other_rank, 0, MPI_COMM_WORLD);
75         }
76     }else{
77         MPI_Recv(buf, sizeof(buf), MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
78         printf("%s\n", buf);
79     }
80
81     /* Tear down the communication infrastructure */

```

```

82     MPI_Finalize();
83 }
84
85 void hello_world_book(){
86     char buf[256];
87     int my_rank, num_procs;
88
89     /* Initialize the infrastructure necessary for communication */
90     MPI_Init(NULL,NULL);
91     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
92     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
93
94     if (my_rank == 0) {
95         printf("Master process of %d processes\n",num_procs);
96
97         for (int other_rank = 1; other_rank < num_procs; other_rank++){
98             MPI_Recv(buf, sizeof(buf), MPI_CHAR, other_rank, 0, MPI_COMM_WORLD,MPI_S
99             printf("%s\n",buf);
100         }
101     }else{
102         sprintf(buf,"Process %d of %d",my_rank,num_procs);
103         MPI_Send(buf, strlen(buf)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
104     }
105
106     /* Tear down the communication infrastructure */
107     MPI_Finalize();
108 }
109
110
111 #define LIST_SIZE 100000000
112 bool merge_sort_recurse(int* buf,int size,int* work);
113 void merge_sort_merge(int* left_buf,int left_size,int* right_buf,int right_size,int*
114 bool gather_tree(int* local_buf,int* local_work,int local_buf_len,int num_procs,int r
115 void merge_sort(){
116     struct timeval timecheck;
117     long start_time,end_time;
118     int my_rank, num_procs;
119     srand(time(NULL));
120
121     MPI_Init(NULL,NULL);
122     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
123     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

```

```

124
125     int local_buf_len = LIST_SIZE / num_procs; // how elements are in normal buffs
126     int buf_extra_elms = LIST_SIZE - local_buf_len*num_procs; // how many extra elms
127     int* local_buf = (int*)malloc(LIST_SIZE*sizeof(int));
128     int* local_work = (int*)malloc(LIST_SIZE*sizeof(int));
129
130     if(my_rank == 0){
131         int* buf = (int*)malloc(LIST_SIZE*sizeof(int));
132         for(int i=0; i<LIST_SIZE; i++)
133             buf[i] = (rand()/(double)(RAND_MAX)) * LIST_SIZE;
134
135         MPI_Barrier(MPI_COMM_WORLD);
136         gettimeofday(&timecheck, NULL);
137         start_time = (long)timecheck.tv_sec * 1000000 + (long)timecheck.tv_usec;
138
139         MPI_Scatter(buf, local_buf_len, MPI_INT,
140                     local_buf, local_buf_len, MPI_INT, 0,
141                     MPI_COMM_WORLD);
142         for(int x=0; x<buf_extra_elms; x++)
143             local_buf[x+local_buf_len] = buf[x + (local_buf_len*num_procs)];
144         bool swapped =
145         merge_sort_recurse(local_buf, local_buf_len+buf_extra_elms, local_work);
146
147         int extras[buf_extra_elms];
148         for(int x=0; x<buf_extra_elms; x++)
149             extras[x] = local_buf[x+local_buf_len];
150
151         if(swapped){
152             int* temp = local_buf;
153             local_buf = local_work;
154             local_work = temp;
155         }
156         swapped = gather_tree(local_buf, local_work, local_buf_len, num_procs, my_rank);
157         free(buf);
158     }else{
159         MPI_Barrier(MPI_COMM_WORLD);
160         MPI_Scatter(NULL, 0, MPI_INT,
161                     local_buf, local_buf_len, MPI_INT, 0,
162                     MPI_COMM_WORLD);
163         bool swapped =
164         merge_sort_recurse(local_buf, local_buf_len, local_work);
165

```

```

166         if(swapped){
167             int* temp = local_buf;
168             local_buf = local_work;
169             local_work = temp;
170         }
171         swapped = gather_tree(local_buf,local_work,local_buf_len,num_procs,my_rank);
172     }
173
174     MPI_Barrier(MPI_COMM_WORLD);
175
176     if(my_rank == 0){
177         gettimeofday(&timecheck, NULL);
178         end_time = (long)timecheck.tv_sec * 1000000 + (long)timecheck.tv_usec;
179
180         printf("Time Taken : %ld\n",end_time - start_time);
181     }
182
183     free(local_buf);
184     free(local_work);
185
186     MPI_Finalize();
187 }
188 bool merge_sort_recurse(int* buf,int size,int* work){
189     if(size==1){
190         // *work = *buf;
191         return false;
192     }
193     if(size==2){
194         if(buf[0] > buf[1]){
195             int temp = buf[1];
196             buf[0] = buf[1];
197             buf[1] = temp;
198         }
199         // work[0] = buf[0];
200         // work[1] = buf[1];
201         //printf(" %d %d\n\n",work[0],work[1]);
202         return false;
203     }
204     // recursive part
205     int half = size/2;
206     bool swap_left=false,swap_right=false;
207     swap_left =

```

```

208     merge_sort_recurse(buf, half, work);
209     swap_right =
210     merge_sort_recurse(buf+half, size-half, work+half); // subtraction for odd size
211     if(swap_left != swap_right){
212         if(swap_right)
213             for(int i=0; i<half; i++)
214                 work[i] = buf[i];
215         else
216             for(int i=0; i<half; i++)
217                 buf[i] = work[i];
218     }
219 }
220 if(swap_right){
221     int* temp = work;
222     work = buf;
223     buf = temp;
224 }
225 merge_sort_merge(buf, half, buf+half, size-half, work);
226
227 return !swap_right;
228 }
229
230 void merge_sort_merge(int* left_buf, int left_size, int* right_buf, int right_size, int*
231 while(left_size && right_size){
232     if(*left_buf <= *right_buf){
233         *dest = *left_buf;
234         left_size--;
235         left_buf++;
236     }else{
237         *dest = *right_buf;
238         right_size--;
239         right_buf++;
240     }
241     dest++;
242 }
243 while(left_size){
244     *dest = *left_buf;
245     left_size--;
246     left_buf++;
247     dest++;
248 }
249 while(right_size){

```



```

250         *dest = *right_buf;
251         right_size--;
252         right_buf++;
253         dest++;
254     }
255
256     return;
257 }
258
259 //0 1 2 3 4 5 6 7 8 9
260 //0 1 2 3 4
261 //0 1 2
262 //0    2
263
264 //0 1 2 3 4 5 6 7 8
265 //0 1 2 3          8
266 //0 1              8
267 //0                8
268
269 //0 1 2 3 4 5 6 7
270 //0 1 2 3
271 //0 1
272 //0
273
274 //0 1 2 3 4 5 6      h=3
275 //0 1 2          6   h=1
276 //0 1  2        6   h=1
277 //0    2        6   h=0
278 bool gather_tree(int* local_buf, int* local_work, int local_buf_len, int num_procs, int m
279     bool swapped = false;
280
281     int half = num_procs/2;
282     int extra_proc=0, extra_size;
283     while(half){
284         if(my_rank<half){
285             MPI_Recv(local_buf+local_buf_len, local_buf_len, MPI_INT, my_rank+half, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
286             merge_sort_merge(local_buf, local_buf_len, local_buf+local_buf_len, local_buf_len);
287         } else if(my_rank<(2*half)){
288             MPI_Send(local_buf, local_buf_len, MPI_INT, my_rank-half, 0, MPI_COMM_WORLD);
289             return swapped; //we gave our work to someone else, so we are done
290         } else{
291             // printf("%d is extra\n", my_rank);

```

```

292         // if(extra_proc==0){
293         //     extra_proc = my_rank;
294         //     extra_size = local_buf_len;
295         // }elseif
296         //     if(extra_proc == my_rank){
297         //         MPI_Send(local_buf, extra_size, MPI_INT, my_rank-half, 0, MPI_COMM_WORLD);
298         //         return swapped; //we gave our work to someone else, so we are done
299         //     }elseif
300         //         MPI_Recv(local_buf+local_buf_len, local_buf_len, MPI_INT, my_rank+half, 0, MPI_COMM_WORLD);
301         //         merge_sort_merge(local_buf, local_buf_len, local_buf+local_buf_len, local_buf_len);
302         //     }
303         //     extra_proc = 0;
304         // }
305     }
306     half/=2;
307     local_buf_len*=2;
308     swapped = !swapped;
309     int* temp = local_buf;
310     local_work = local_buf;
311     local_buf=temp;
312 }
313 return swapped;
314 }

```

C pthread2.cpp

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <unistd.h>
5  #include <sys/time.h>
6  #include <pthread.h>
7
8  #define LIST_SIZE 100000000
9  // #define LIST_SIZE 10
10 #define NUM_THREADS 2
11
12 void merge_sort();
13
14 int main(){
15     merge_sort();
16     return 0;
17 }
18
19 struct list_item{list_item* next;int *left,*right,left_size,right_size,*work;};
20 pthread_mutex_t queue_mutex;
21 pthread_barrier_t sync_barrier;
22 struct list_item *list_front=NULL,*list_back=NULL;
23 void add_work_item(int*left,int*right,int left_size,int right_size,int*work){
24     //printf("%d %d\n",left_size,right_size);
25     //pthread_mutex_lock(&queue_mutex);
26     struct list_item *prev = list_back;
27     list_back = new struct list_item;
28     list_back->next=NULL;
29     list_back->left=left;
30     list_back->left_size=left_size;
31     list_back->right=right;
32     list_back->right_size=right_size;
33     list_back->work=work;
34     if(list_front == NULL){
35         list_front = list_back;
36     }
37     if(prev != NULL)
38         prev->next = list_back;
39     //pthread_mutex_unlock(&queue_mutex);
```

```

40 }
41 struct list_item* get_next_work(){
42     pthread_mutex_lock(&queue_mutex);
43     struct list_item* temp = list_front;
44     if(temp != NULL)
45         list_front = list_front->next;
46     pthread_mutex_unlock(&queue_mutex);
47     return temp;
48 }
49 // bool init_work_list(int*buf,int size,int*work){
50 //     if(size<=2){
51 //         add_work_item(buf,size,work);
52 //         return false;
53 //     }
54 //     int half = size/2;
55 //     bool swap_left = init_work_list(buf,half,work);
56 //     bool swap_right = init_work_list(buf+half,size-half,work+half);
57 //     if(swap_left != swap_right){
58 //         if(swap_right)
59 //             for(int i=0; i<half; i++)
60 //                 work[i] = buf[i];
61 //         else
62 //             for(int i=0; i<half; i++)
63 //                 buf[i] = work[i];
64 //     }
65 //     if(swap_right){
66 //         int* temp = work;
67 //         work = buf;
68 //         buf = temp;
69 //     }
70 // }
71 //
72 //     add_work_item(buf,half,work);
73 //     add_work_item(buf+half,size-half,work+half);
74 //     //add_work_item(NULL,0,NULL);
75 //
76 //     return !swap_right;
77 // }
78
79 bool init_work_2(int*buf,int size,int*work){
80     bool swapped = false;
81     int step=2;

```

```

82  //handle the base cases first
83  for(int x=0; x<size; x+=2){
84      if(x+2<=size){
85          add_work_item(buf+x,NULL,2,0,work+x);
86      }else
87          add_work_item(buf+x,NULL,1,0,work+x);
88  }
89  // sync all the threads
90  for(int x=0; x<NUM_THREADS; x++)
91      add_work_item(NULL,NULL,0,0,NULL);
92
93  //now the merging part
94  while(step<size){
95      for(int curt_step=0; curt_step<size; curt_step+=2*step){
96          if(curt_step+2*step<size){
97              //normal full merge
98              add_work_item(buf+curt_step,buf+curt_step+step,step,step,work+curt_step);
99          }else if(curt_step+step<size){
100              //left full size, right partial size
101              int partial = size - (curt_step+step);
102              add_work_item(buf+curt_step,buf+curt_step+step,step,partial,work+curt_step);
103          }else{
104              //left partial size
105              int partial = size - curt_step;
106              add_work_item(buf+curt_step,NULL,partial,0,work+curt_step);
107          }
108      }
109      // sync all the threads
110      for(int x=0; x<NUM_THREADS; x++)
111          add_work_item(NULL,NULL,0,0,NULL);
112
113      step*=2;
114      swapped = !swapped;
115      int*temp = buf;
116      buf=work;
117      work=temp;
118  }
119  return swapped;
120 }
121
122
123 void* worker_thread(void*);

```

```

124 void merge_sort_sort(int* buf,int size,int* work);
125 void merge_sort_merge(int* left_buf,int left_size,int* right_buf,int right_size,int*
126 void merge_sort(){
127     struct timeval timecheck;
128     long start_time,end_time;
129     srand(time(NULL));
130     pthread_mutex_init(&queue_mutex,NULL);
131     pthread_barrier_init(&sync_barrier,NULL,NUM_THREADS);
132
133     int* buf = (int*)malloc(LIST_SIZE*sizeof(int));
134     int* work = (int*)malloc(LIST_SIZE*sizeof(int));
135     for(int i=0;i<LIST_SIZE;i++)
136         buf[i] = (rand()/(double)(RAND_MAX)) * LIST_SIZE;
137
138     gettimeofday(&timecheck, NULL);
139     start_time = (long)timecheck.tv_sec * 1000000 + (long)timecheck.tv_usec;
140     bool is_backwards =
141     //init_work_list(buf,LIST_SIZE,work);
142     init_work_2(buf,LIST_SIZE,work);
143     {
144         pthread_t handles[NUM_THREADS];
145         for(int x=0; x<NUM_THREADS; x++)
146             pthread_create(handles+x,NULL,worker_thread,NULL);
147         for(int x=0; x<NUM_THREADS; x++)
148             pthread_join(handles[x],NULL);
149     }
150     //merge_sort_recurse(buf,LIST_SIZE,work);
151     gettimeofday(&timecheck, NULL);
152     end_time = (long)timecheck.tv_sec * 1000000 + (long)timecheck.tv_usec;
153
154     // for(int x=0; x<LIST_SIZE; x++)
155     //     if(is_backwards)
156     //         printf("%d ", work[x]);
157     //     else
158     //         printf("%d ", buf[x]);
159     // printf("\n");
160     printf("Time Taken in us : %ld\n",end_time - start_time);
161     free(buf);
162     free(work);
163 }
164 // void merge_sort_sort(int*left,int*right,int size,int* work){
165 //     if(size==1){

```

```

166 //         return;
167 //     }
168 //     if(size==2){
169 //         if(buf[0] > buf[1]){
170 //             int temp = buf[1];
171 //             buf[0] = buf[1];
172 //             buf[1] = temp;
173 //         }
174 //         return;
175 //     }
176 //     int half = size/2;
177 //     merge_sort_merge(buf, half, buf+half, size-half, work);
178 // }
179 void merge_sort_merge(int* left_buf, int left_size, int* right_buf, int right_size, int*
180     if(left_size==1 && right_buf == NULL){
181         // *dest = *left_buf;
182         return;
183     }
184     if(left_size==2 && right_buf == NULL){
185         if(left_buf[0] > left_buf[1]){
186             int temp = left_buf[1];
187             left_buf[0] = left_buf[1];
188             left_buf[1] = temp;
189         }
190         // dest[0]=left_buf[0];
191         // dest[1]=left_buf[1];
192         return;
193     }
194     while(left_size && right_size){
195         if(*left_buf <= *right_buf){
196             *dest = *left_buf;
197             left_size--;
198             left_buf++;
199         }else{
200             *dest = *right_buf;
201             right_size--;
202             right_buf++;
203         }
204         dest++;
205     }
206     while(left_size){
207         *dest = *left_buf;

```

```

208         left_size--;
209         left_buf++;
210         dest++;
211     }
212     while(right_size){
213         *dest = *right_buf;
214         right_size--;
215         right_buf++;
216         dest++;
217     }
218
219     return;
220 }
221
222 void* worker_thread(void*){
223     auto segment = get_next_work();
224     while(segment != NULL){
225         if(segment->left!=NULL)
226             merge_sort_merge(segment->left,segment->left_size,segment->right,segment->right_size);
227         else // sync command
228             pthread_barrier_wait(&sync_barrier);
229         segment = get_next_work();
230     }
231 }

```


D openmp.cpp

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <unistd.h>
5  #include <sys/time.h>
6  #include <omp.h>
7
8  #define LIST_SIZE 100000000
9  // #define LIST_SIZE 10
10 #define NUM_THREADS 8
11
12 void merge_sort();
13
14 int main(){
15     merge_sort();
16     return 0;
17 }
18
19 struct list_item{list_item* next;int *left,*right,left_size,right_size,*work;};
20 struct list_item *list_front=NULL,*list_back=NULL;
21 void add_work_item(int*left,int*right,int left_size,int right_size,int*work){
22     //printf("%d  %d\n",left_size,right_size);
23     //pthread_mutex_lock(&queue_mutex);
24     struct list_item *prev = list_back;
25     list_back = new struct list_item;
26     list_back->next=NULL;
27     list_back->left=left;
28     list_back->left_size=left_size;
29     list_back->right=right;
30     list_back->right_size=right_size;
31     list_back->work=work;
32     if(list_front == NULL){
33         list_front = list_back;
34     }
35     if(prev != NULL)
36         prev->next = list_back;
37     //pthread_mutex_unlock(&queue_mutex);
38 }
39 struct list_item* get_next_work(){
```

```

40     struct list_item* temp = list_front;
41     if(temp != NULL)
42         list_front = list_front->next;
43     return temp;
44 }
45 // bool init_work_list(int*buf,int size,int*work){
46 //     if(size<=2){
47 //         add_work_item(buf,size,work);
48 //         return false;
49 //     }
50 //     int half = size/2;
51 //     bool swap_left = init_work_list(buf,half,work);
52 //     bool swap_right = init_work_list(buf+half,size-half,work+half);
53 //     if(swap_left != swap_right){
54 //         if(swap_right)
55 //             for(int i=0; i<half; i++)
56 //                 work[i] = buf[i];
57 //         else
58 //             for(int i=0; i<half; i++)
59 //                 buf[i] = work[i];
60 //     }
61 //     if(swap_right){
62 //         int* temp = work;
63 //         work = buf;
64 //         buf = temp;
65 //     }
66 // }
67
68 //     add_work_item(buf,half,work);
69 //     add_work_item(buf+half,size-half,work+half);
70 //     //add_work_item(NULL,0,NULL);
71
72 //     return !swap_right;
73 // }
74
75 bool init_work_2(int*buf,int size,int*work){
76     bool swapped = false;
77     int step=2;
78     //handle the base cases first
79     for(int x=0; x<size; x+=2){
80         if(x+2<=size){
81             add_work_item(buf+x,NULL,2,0,work+x);

```

```

82         }else
83             add_work_item(buf+x,NULL,1,0,work+x);
84     }
85     // sync all the threads
86     for(int x=0; x<NUM_THREADS; x++)
87         add_work_item(NULL,NULL,0,0,NULL);
88
89     //now the merging part
90     while(step<size){
91         for(int curt_step=0; curt_step<size; curt_step+=2*step){
92             if(curt_step+2*step<size){
93                 //normal full merge
94                 add_work_item(buf+curt_step,buf+curt_step+step,step,step,work+curt_step);
95             }else if(curt_step+step<size){
96                 //left full size, right partial size
97                 int partial = size - (curt_step+step);
98                 add_work_item(buf+curt_step,buf+curt_step+step,step,partial,work+curt_step);
99             }else{
100                 //left partial size
101                 int partial = size - curt_step;
102                 add_work_item(buf+curt_step,NULL,partial,0,work+curt_step);
103             }
104         }
105         // sync all the threads
106         for(int x=0; x<NUM_THREADS; x++)
107             add_work_item(NULL,NULL,0,0,NULL);
108
109         step*=2;
110         swapped = !swapped;
111         int*temp = buf;
112         buf=work;
113         work=temp;
114     }
115     return swapped;
116 }
117
118
119 void* worker_thread(void*);
120 void merge_sort_sort(int* buf,int size,int* work);
121 void merge_sort_merge(int* left_buf,int left_size,int* right_buf,int right_size,int* work);
122 void merge_sort(){
123     struct timeval timecheck;

```

```

124     long start_time,end_time;
125     srand(time(NULL));
126
127     int* buf = (int*)malloc(LIST_SIZE*sizeof(int));
128     int* work = (int*)malloc(LIST_SIZE*sizeof(int));
129     for(int i=0;i<LIST_SIZE;i++)
130         buf[i] = (rand()/(double)(RAND_MAX)) * LIST_SIZE;
131
132     gettimeofday(&timecheck, NULL);
133     start_time = (long)timecheck.tv_sec * 1000000 + (long)timecheck.tv_usec;
134     bool is_backwards =
135         //init_work_list(buf,LIST_SIZE,work);
136         init_work_2(buf,LIST_SIZE,work);
137         #pragma omp parallel num_threads(NUM_THREADS)
138         worker_thread(NULL);
139         //merge_sort_recurse(buf,LIST_SIZE,work);
140     gettimeofday(&timecheck, NULL);
141     end_time = (long)timecheck.tv_sec * 1000000 + (long)timecheck.tv_usec;
142
143     // for(int x=0; x<LIST_SIZE; x++)
144     //     if(is_backwards)
145     //         printf("%d ", work[x]);
146     //     else
147     //         printf("%d ", buf[x]);
148     // printf("\n");
149     printf("Time Taken in us : %ld\n",end_time - start_time);
150     free(buf);
151     free(work);
152 }
153 // void merge_sort_sort(int*left,int*right,int size,int* work){
154 //     if(size==1){
155 //         return;
156 //     }
157 //     if(size==2){
158 //         if(buf[0] > buf[1]){
159 //             int temp = buf[1];
160 //             buf[0] = buf[1];
161 //             buf[1] = temp;
162 //         }
163 //         return;
164 //     }
165 //     int half = size/2;

```

```

166 //      merge_sort_merge(buf, half, buf+half, size-half, work);
167 // }
168 void merge_sort_merge(int* left_buf, int left_size, int* right_buf, int right_size, int*
169     if(left_size==1 && right_buf == NULL){
170         // *dest = *left_buf;
171         return;
172     }
173     if(left_size==2 && right_buf == NULL){
174         if(left_buf[0] > left_buf[1]){
175             int temp = left_buf[1];
176             left_buf[0] = left_buf[1];
177             left_buf[1] = temp;
178         }
179         // dest[0]=left_buf[0];
180         // dest[1]=left_buf[1];
181         return;
182     }
183     while(left_size && right_size){
184         if(*left_buf <= *right_buf){
185             *dest = *left_buf;
186             left_size--;
187             left_buf++;
188         }else{
189             *dest = *right_buf;
190             right_size--;
191             right_buf++;
192         }
193         dest++;
194     }
195     while(left_size){
196         *dest = *left_buf;
197         left_size--;
198         left_buf++;
199         dest++;
200     }
201     while(right_size){
202         *dest = *right_buf;
203         right_size--;
204         right_buf++;
205         dest++;
206     }
207

```

```

208     return;
209 }
210
211 void* worker_thread(void*){
212     struct list_item* segment;
213     #pragma omp critical
214     segment = get_next_work();
215     while(segment != NULL){
216         if(segment->left!=NULL)
217             merge_sort_merge(segment->left,segment->left_size,segment->right,segment->right_size);
218         else{ // sync command
219             #pragma omp barrier
220             ;}
221         ;
222         #pragma omp critical
223         segment = get_next_work();
224     }
225 }

```